ChatGPT

Epic OAuth & Multi-Provider Refactoring Considerations

1. Epic OAuth and Practitioner Role Requirement

Epic's FHIR User Context: Epic's SMART on FHIR OAuth requires that the **authenticated user correspond to a FHIR resource** (e.g. a Practitioner, Patient, or RelatedPerson) in the Epic system. In provider-facing workflows, the user's identity is represented as a **Practitioner** resource [1]. If a user cannot be represented by a FHIR resource, the `fhirUser` OpenID scope is essentially unsupported [2]. In practice, this means a **non-clinical user (e.g. a practice manager)** may not be able to complete the OAuth flow with the `fhirUser` scope because they have no Practitioner record in FHIR. The Epic sandbox likely uses test clinician accounts (Practitioners), which is why OAuth has only been successful with those users.

Impact on Current Design: In your current implementation, you assumed an **organization admin** (e.g. practice manager) could perform the OAuth handshake and share the token with sub-users. However, given Epic's requirements, **only a user with a Practitioner record can authorize those** `user/*` **scopes** and yield a usable token. This forces your "admin" user to also be a Practitioner in Epic, which is often not true for real clinics. It also complicates onboarding – originally your admin just needed to enter a Client ID, secret, and an email, but now the admin might also need **a provider's Epic credentials** to complete OAuth. This is a critical design mismatch.

Potential Solutions – Separate Roles vs. Provider OAuth: You have two main options to address this:

- **a. Introduce Distinct "Epic Integration Admin" Role:** Separate the concerns of business administration and Epic connectivity. A **"Business Admin"** user (office manager, billing staff, etc.) can manage billing, user accounts, and settings, **but cannot directly authorize Epic**. Instead, have an **"Epic Integration Admin"** role (or simply require a provider user) for connecting to Epic. In practice, this could mean the Business Admin invites or designates **a Practitioner account to perform the OAuth login**. The provider (or another user who *is* a Practitioner in Epic) would log in via the OAuth flow to authorize the app. The obtained access/refresh token can then be stored at the organization level. This way, the practice manager doesn't need to use their own (non-provider) credentials – they leverage a provider's credentials for the integration. This approach aligns with Epic's expectations (since a Practitioner is authorizing), but still allows non-clinical staff to handle other setup aspects.

- **b. Require Each Provider to Authorize Separately:** In a fully provider-centric model, **each provider user in your system does their own OAuth connection to Epic** using their individual Epic login. Each provider's token would grant access to that provider's data (patients, appointments, etc.). Sub-users could then view data **per provider** based on whose token is used. This ensures every token maps to a Practitioner (satisfying Epic's requirement) and cleanly links data to the correct provider. The downside is onboarding friction – **every provider must go through the OAuth process** (for a large practice this is extra work). You would also need to manage multiple tokens (one per provider) and refresh them (your `offline_access` scope allows background refresh).

Recommendation: Given the constraints, it's often best to **separate the roles** so that Epic OAuth is done by a Practitioner. If an organization has only one provider, that provider can simply be the admin and do it all. For multi-provider groups, you might implement a workflow where an admin user can **"Assign an Epic Connection"** to a particular provider: e.g. the practice manager clicks "Connect to Epic" for Dr. Smith, which prompts Dr. Smith (or the admin, if they have Dr. Smith's login handy) to complete OAuth. Once connected, the token for Dr. Smith is stored, and **all Dr. Smith's data** can sync. Repeat for other providers as needed. This way, **business admins handle coordination** but **providers handle actual authorization**. It's a bit of a hybrid of (a) and (b).

(Note: In theory, Epic could support a backend "system" credential flow using client assertions and system scopes instead of user-level scopes. You even have code in place for JWT client assertions. However, Epic's App Orchard typically requires special setup for system-level API access, and your listed scopes are `user/*` *and* `openid/fhirUser` *which imply a user context. So, sticking with user/Practitioner context is the practical route for now.)*

2. Billing Model – Per-Organization vs. Per-Provider

Your current billing is one subscription **per organization (per Client ID)**, but if you shift to a provider-centric integration, you should reassess how value is measured:

- **Per-Provider Pricing:** If each provider is an independent "unit of work" (with their own schedule, patients, and Epic connection), a **per-provider pricing model** might make sense. This scales revenue with usage – a clinic with 5 providers would pay roughly 5× more than a solo practitioner, reflecting the greater benefit and resources used. It also lets you price **per provider license** which some clinics expect for provider-facing software.

- **Per-Organization Flat Pricing:** A flat fee per org (regardless of provider count) is simpler for customers to understand and for you to manage billing. It encourages clinics to onboard all their providers without worrying about cost per head. However, a single flat rate might undervalue large multi-provider clinics (who will consume more API calls, support, etc. on your end). If you stick to per-org pricing, you might consider tiered plans (e.g. an org with up to 3 providers vs. 10 providers, etc.) to ensure large practices pay more for the additional value they get.

Impact on Value Proposition: Think about how you present the product's value. If **each provider's workflows are being improved** (e.g. each doctor gets automated screening and scheduling integration), charging per provider highlights that value clearly. On the other hand, if the product is pitched as an organizational tool (e.g. improving clinic operations overall), a single org fee might be more palatable. You'll need to decide which approach aligns with your sales strategy. Many B2B health software companies use per-provider pricing for clinician-facing features, but you can also do a hybrid (e.g. base fee per practice + add-on fee per additional provider).

3. Admin–Provider Relationship Model

With the above changes, you'll need to refactor your user and data model to support **multiple providers per organization** and flexible admin roles. Key considerations:

- **One Admin to Many Providers:** It should be possible for a single "Business Admin" user (e.g. a practice manager) to manage settings and data for **multiple providers in the same organization**. In a group practice, one office manager might oversee 5 doctors – your system should allow that one admin account to have access to all 5 providers' data (assuming the providers' tokens are connected). Typically, if users are tied to an `Organization` entity, an admin user with org-level privileges can see all providers under that org.

- **Multiple Admins per Provider/Org:** Conversely, you may have scenarios where **a provider and an office manager** both need access to the provider's data. Or a clinic might have two admins (e.g. front-desk scheduler and billing specialist) who both manage the same set of providers. Your model should allow **multiple admin users for one organization**, and by extension, those admins can manage overlapping providers. This is essentially a **many-to-many relationship** between admin users and provider profiles within the org. For simplicity, you might implement it as all admins of an org can manage all providers of that org (common in small clinics). If you need more granularity (e.g. each admin is assigned to specific providers), you can create an explicit mapping table between admin accounts and provider accounts.

- **Providers as Users vs. Separate Entity:** Decide how to represent providers in your system. Likely you'll introduce a **Provider model** (with fields like name, specialty, Epic Practitioner ID, etc.) which is linked to the Organization. Each Provider might also correspond to a user account (if providers log into your app). It's okay for a user to have dual roles – e.g. Dr. Alice is a **provider-user** (with her Practitioner ID and Epic token) *and* she might also be marked as an admin if you want her to manage other users or billing. Meanwhile, an office manager is an **admin-user** without a provider record. This flexibility covers cases like a solo practitioner (one person is both admin and provider) as well as group practices (admins and providers are separate people).

Designing the schema as **Organization – Provider – User**, with junction tables or role flags, will give you the needed flexibility. For example, you could have an `Organization` table, a `Provider` table (foreign key to Organization), and a `User` table. Users have roles (admin or provider or both) and if a user is a provider, they link to one Provider record. Admin users might have a foreign key to Organization (to scope their access) and possibly a relation to providers they manage (if not all in org). This approach supports one-to-many and many-to-many as needed.

4. Epic Practitioner ID Mapping (Provider Linking)

Once you have local Provider records, you **must link each to the correct Epic Practitioner**. This is crucial for filtering FHIR data (e.g. pulling only Dr. Smith's appointments). Here's how to tackle it:

- **Capture Practitioner ID via OAuth:** When a provider user authorizes via Epic OAuth, the resulting **ID Token** (or the UserInfo endpoint) will include a `fhirUser` claim if scopes `openid` and `fhirUser` were requested. In Epic's system, for a provider login this `fhirUser` claim is an

absolute URL of their Practitioner resource (e.g. `.../Practitioner/12345`) [1] [3]. You can parse that to get the Epic Practitioner ID. During onboarding, as soon as Dr. Smith completes OAuth, store the `Practitioner/12345` ID in Dr. Smith's Provider record in your database. This auto-links the local provider to their Epic identity without the admin having to manually input it.

- **Validating or Discovering IDs:** If for some reason you need to obtain a practitioner's Epic ID outside of an OAuth login, Epic's FHIR API allows you to **search for practitioners by identifier or name**. For example, you can call `GET /Practitioner?identifier=<system>|<ID>` (where the identifier could be an NPI, provider number, or other known ID) [4]. If your onboarding process knows the provider's NPI or their Epic user ID, you could query Epic for the Practitioner resource. This could be used to validate that the provider exists in the Epic system and to retrieve the correct Practitioner resource ID. Another approach is to let the user select from a list: for instance, query `/Practitioner?name=Smith` to get a list of Dr. Smiths and have the admin confirm which one is their provider (this might be needed if an admin is setting up providers without each logging in, but OAuth flow is the more direct method).

- **Data Fetch Using Practitioner ID:** With the Epic Practitioner ID linked, your sync logic can use it to fetch resources scoped to that provider. For example, to sync appointments you might call Epic's Appointment search filtered by `Practitioner/{id}` as the actor or provider parameter. The token obtained via that provider's OAuth will usually be limited to data that provider is permitted to see (which generally includes their own appointments, encounters, etc.). Linking IDs correctly ensures you're pulling **only the relevant records**.

(Side note: it's wise to store not just the ID but also perhaps the practitioner's name or NPI for cross-checking. And during the OAuth callback, you might verify the token's scopes and that the `fhirUser` claim indeed contains "Practitioner/...". If the claim were instead "Patient/...", you'd know a patient account was used by mistake.)

5. Migrating Existing Data to Provider Model

Since your system initially had no Provider differentiation (all patients/appointments tied to the org as a whole), you need a migration plan. Fortunately, you mentioned **no live production clients yet (only sandbox testing data)**, which makes this easier. You have a couple of options:

- **Automatic "Default Provider" Creation:** For each existing Organization in your database, create a single Provider entry (e.g. "Default Provider" or use the org's name as the provider name). Migrate all existing patients, appointments, and related records to be associated with this provider. This way, nothing is lost – every piece of data now has a provider reference. In a migration script, you could: for each Org without providers, create one Provider, and update all that org's patient/appointment records to point to that provider's ID. This approach is simplest and requires no manual intervention. It basically preserves the old behavior (each org acts like it had one provider).

- **Manual Assignment (if needed):** If you did have some data that needed splitting by provider, a manual approach would be to leave data unassigned and have an admin user assign each patient or appointment to a provider after you deploy the new model. However, this is **not** ideal to impose on users, especially since you don't actually have real multi-provider data yet. Given your early stage (no live clients), it's better to handle it automatically and avoid any confusion.

After migration, you can allow organizations to add real providers through your UI. For the existing “Default Provider” entries, an admin could either repurpose it (rename it to the actual doctor’s name if it was a single-doctor clinic) or simply delete it once they create their actual provider profiles and re-link any data as needed. Since in testing you likely only had one “provider” concept per org, the default aligns with that.

Conclusion: By addressing the above concerns, you’ll refactor the system to be robust for real-world Epic integrations. In summary, plan to **have OAuth performed by actual Epic Practitioner users** (meeting Epic’s `fhirUser` requirement [2]), redesign your data model to support **multiple providers per org with flexible admin oversight**, and adjust your business logic (and possibly pricing) to treat providers as first-class entities. This will position your application to handle scenarios like multi-specialty clinics, while staying compliant with Epic’s OAuth workflow and making the onboarding smoother for both admins and providers.

Sources: Epic FHIR documentation confirms that the `fhirUser` OpenID Connect claim will reference a Practitioner resource for provider-authenticated sessions [1], and it notes that if a user can’t be mapped to a FHIR resource, the `fhirUser` scope isn’t supported [2]. Additionally, Epic’s API allows searching for Practitioner resources by identifiers (such as NPI) to help link local providers to Epic’s Practitioner IDs [4]. These details informed the recommendations above.

---

[1] [3] [4] Documentation - Epic on FHIR
https://fhir.epic.com/Documentation

[2] App Launch: Scopes and Launch Context - SMART App Launch v2.2.0
https://build.fhir.org/ig/HL7/smart-app-launch/scopes-and-launch-context.html